Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

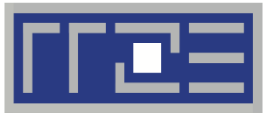
Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 12: MPI I/O

High Performance Computing

Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming



MPI Input/Output

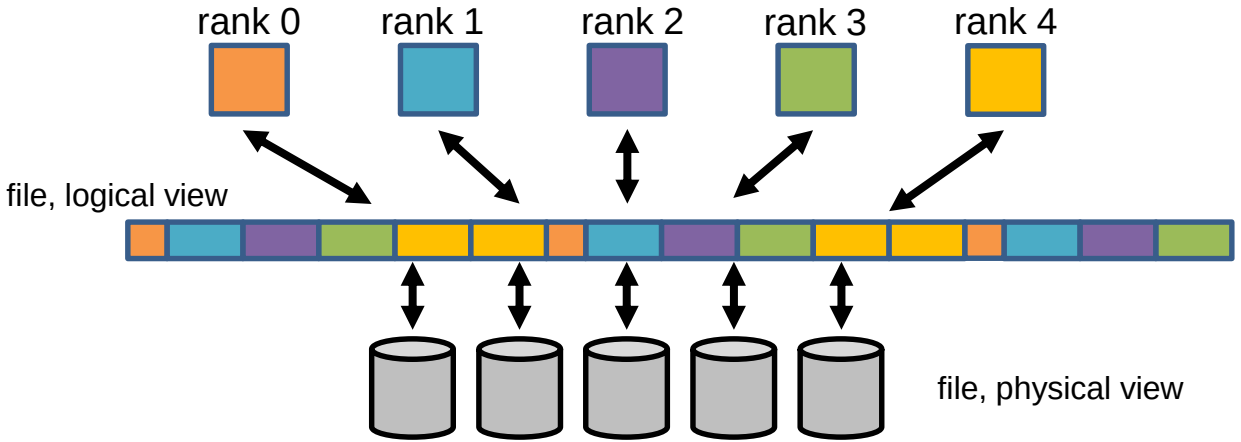
Why MPI I/O?

Many parallel applications need …

- coordinated parallel access to a file by a group of processes,
- simultaneous access to a file,
- non-contiguous access to pieces of the file by many processes,

i.e., the data may be distributed amongst the processes according to a partitioning scheme.

And of course it should be efficient!



MPI I/O features

- MPI I/O Provides a high-level interface to support
  - data file partitioning among processes
  - transfer global data between memory and files (“collective” I/O)
  - asynchronous transfers
  - strided access

- MPI derived data types are used to specify common data access patterns for maximum flexibility and expressiveness

MPI I/O: principles

- MPI file contains elements of a single MPI data type (etype)
- The file is partitioned among processes using an access template (filetype)
- All file accesses transfer to/from a contiguous or non-contiguous user buffer (MPI data type)
- Several different ways of reading/writing data:
  - non-blocking / blocking
  - collective / individual
  - individual / shared file pointers, explicit offsets

- Automatic data conversion in heterogeneous systems
- File interoperability with external representation

Opening & closing files

File open

```
int MPI_File_open(MPI_Comm comm, const char *filename,
                  int amode, MPI_Info info,
                  MPI_File *fh);
```

- Collective call by all processes which are part of **comm**
- **filename** can be different, but must point to the same file
- **amode** describes access mode (see next slide)
- **info** object, can be **MPI_INFO_NULL** (see later)
- **fh** represents the file handle, to which **comm** and the view (see later) are associated

- Process-local file I/O is possible by specifying **MPI_COMM_SELF** as **comm**

File access modes

Access mode	Description	
`MPI_MODE_RDONLY`	read only	one of these is required
`MPI_MODE_RDWR`	read and write	
`MPI_MODE_WRONLY`	write only	
`MPI_MODE_CREATE`	create if it does not exist	
`MPI_MODE_EXCL`	error if file exists	
`MPI_MODE_DELETE_ON_CLOSE`	file is deleted when closed	
`MPI_MODE_UNIQUE_OPEN`	file is not concurrently opened by anybody else	
`MPI_MODE_SEQUENTIAL`	only sequential access will occur (`MPI_File_read/write_shared` is allowed)	
`MPI_MODE_APPEND`	all file pointers are located at the end of the file	

Flags can be or'ed together, e.g., **`MPI_MODE_WRONLY | MPI_MODE_APPEND`**

File open

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, filename,
  MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
...
```

- All processes in MPI_COMM_WORLD open the file collectively
- Also possible to open file with only one process:

```
if (rank == 0) {
 MPI_File fh;
 MPI_File_open(MPI_COMM_SELF, filename,
   MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
...
}
```

File close

```
int MPI_File_close(MPI_File *fh);
```

- Collective call by all processes in the communicator the file was opened in
- File state is synchronized, i.e., all data is transferred to disk storage
- File handle **`fh`** is set to **`MPI_FILE_NULL`**
- File is deleted if **`MPI_MODE_DELETE_ON_CLOSE`** was part of access mode
- All outstanding nonblocking requests & split collectives associated with **`fh`** must have been completed

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, …, &fh);
...
MPI_File_close(&fh);
```

Info objects

Info objects I

- Opaque object, storing key/value pairs
- Often used to provide system-specific information
  - via **`info`** argument in function calls
  - for MPI I/O, process management, memory allocation, …
- Keys
  - All keys may be ignored
  - MPI defines a set of reserved keys
  - Implementations may provide additional keys
- Keys/values are strings and converted to other types as required
- Use **`MPI_INFO_NULL`** if you do not want to provide additional information

```
MPI_Info info;
```

New, empty object:

```
int MPI_Info_create(
    MPI_Info *info);
```

Add entry to existing object:

```
int MPI_Info_set(
    MPI_Info info,
    const char *key,
    const char *value);
```

Info objects II

- Delete entry from info object

```
int MPI_Info_delete(MPI_Info info, const char *key);
```

- Retrieve value associated with key

```
int MPI_Info_get(MPI_Info info, const char *key,
        int valuelen, char *value, int *flag);
```

  - **`flag = true`**: a value is associated with the key and returned in **`value`**
  - **`flag = false`**: no value associated with the key, **`value`** is unchanged
  - **`valuelen`**: size of the buffer **`value`** points to, if associated value is larger, data is truncated

- Free info object

```
int MPI_Info_free(MPI_Info *info);
```

- Length restriction:
  - keys: **`MPI_MAX_INFO_KEY`**
  - values: **`MPI_MAX_INFO_VAL`**

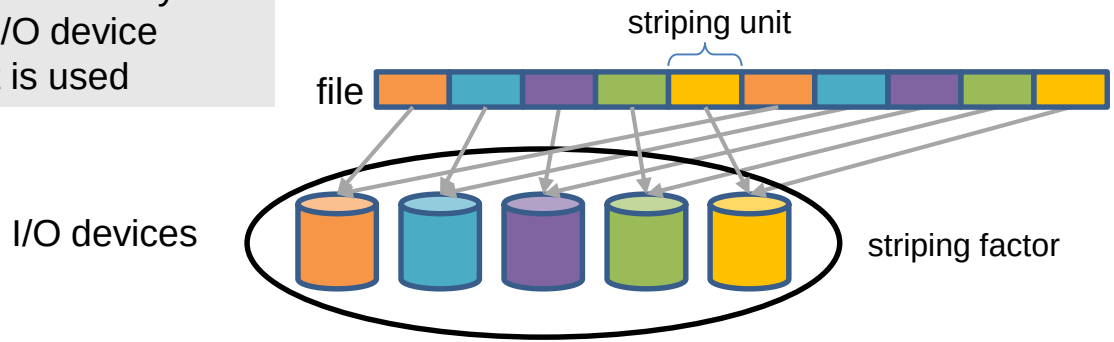
Info objects for striping

Striping:

- relevant only when file is created, i.e. in **`MPI_File_open`**
- must be the same for all processes
- is only a hint

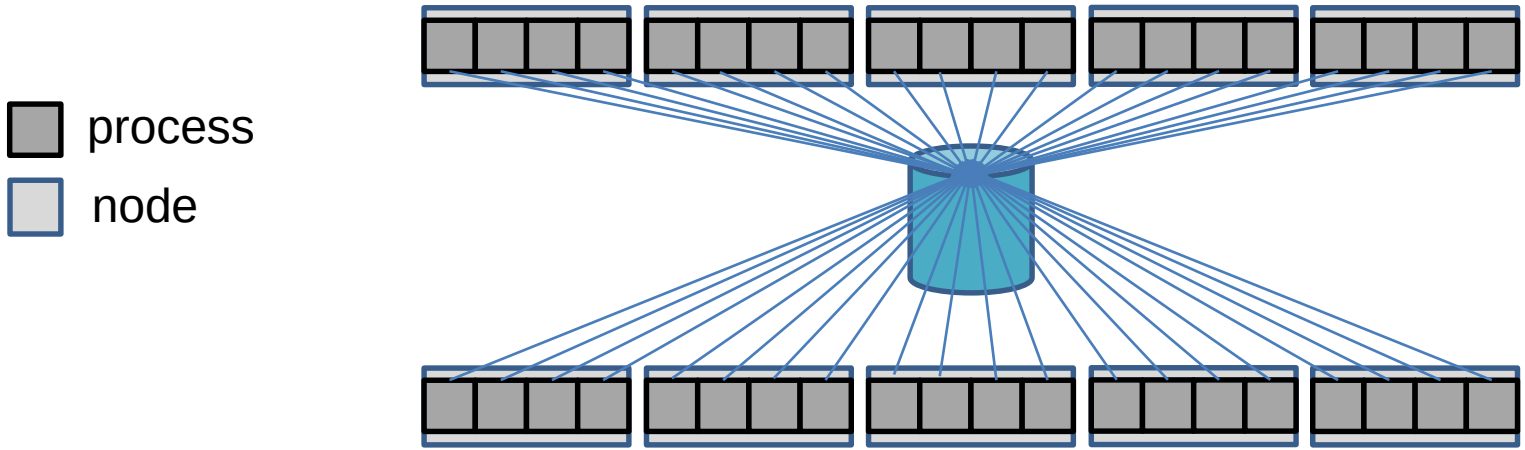
Keys for info object:

`striping_factor`	int	number of I/O devices the file should be striped across
`striping_unit`	int	number of consecutive bytes stored on one I/O device before the next is used



Info objects for collective buffering

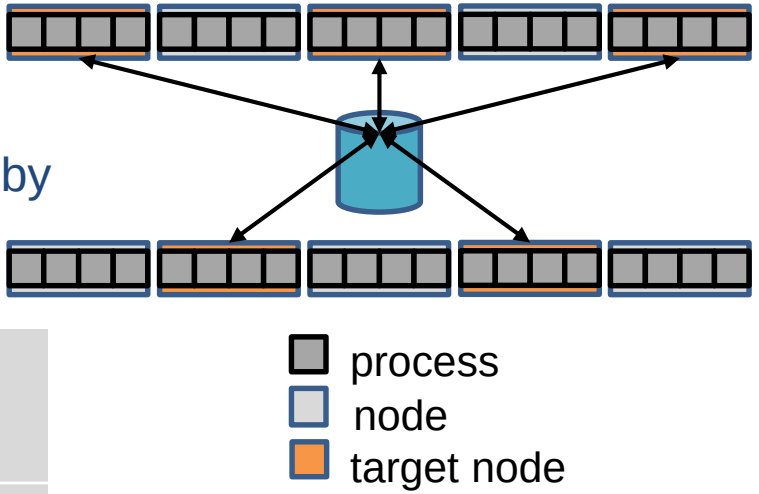
- Each process might access I/O devices
- Can generate high load
- Collective buffering to mitigate this problem



Info objects for collective buffering

Collective buffering

- Optimization for collective accesses
- Access performed on behalf of all processes by some target nodes



Keys for info object:

`collective_buffering`	`bool`	true if application might benefit from collective buffering, false if not
`cb_block_size`	`int`	target nodes access data in chunks of this size
`cb_buffer_size`	`int`	buffer size on target node used for collective buffering, usually a multiple of the block size
`cb_nodes`	`int`	number of target nodes

Info object example

Example: create MPI info object for **MPI_File_open**

```
MPI_Info info;

MPI_Info_create(&info);
// Hint: stripe over 10 I/O devices
MPI_Info_set(info, "striping_factor", "10");
// Hint: enable collective buffering
MPI_Info_set(info, "collective_buffering", "true");
// Hint: use 4 target nodes for buffering
MPI_Info_set(info, "cb_nodes", "4");
...
MPI_File_open(comm, filename, amode, info, &fh);
...
MPI_Info_free(&info);
```

Query info of open file (I)

```
// Error handling omitted for brevity
MPI_Info info;
char keyName[MPI_MAX_INFO_KEY + 1], * value;
int nKeys, nValue, keyDefined;

MPI_File_get_info(fh, &info);
MPI_Info_get_nkeys(info, &nKeys);

for (int i = 0; i < nKeys; ++i) {
        MPI_Info_get_nthkey(info, i, keyName);
        MPI_Info_get_valuelen(info, keyName, &nValue, &keyDefined);

        if (!keyDefined) continue;
        value = (char *)malloc(sizeof(char *) * (nValue + 1));

        MPI_Info_get(info, keyName, nValue, value, &keyDefined);
        printf("info get [%2d] %s: %s\n", i, keyName, value);

        free(value);
}

MPI_Info_free(&info);
```

Query info of open file (II)

RRZE's Meggie cluster, Intel MPI, one process, one file striped over 32 I/O devices on Lustre file system

```
info get [ 0] direct_read: false
info get [ 1] direct_write: false
info get [ 2] romio_lustre_co_ratio: 1
info get [ 3] romio_lustre_coll_threshold: 0
info get [ 4] romio_lustre_ds_in_coll: enable
info get [ 5] cb_buffer_size: 16777216
info get [ 6] romio_cb_read: automatic
info get [ 7] romio_cb_write: automatic
info get [ 8] cb_nodes: 1
info get [ 9] romio_no_indep_rw: false
info get [10] romio_cb_pfr: disable
info get [11] romio_cb_fr_types: aar
info get [12] romio_cb_fr_alignment: 1
info get [13] romio_cb_ds_threshold: 0
info get [14] romio_cb_alltoall: automatic
info get [15] ind_rd_buffer_size: 4194304
info get [16] ind_wr_buffer_size: 524288
info get [17] romio_ds_read: automatic
info get [18] romio_ds_write: automatic
info get [19] cb_config_list: *:1
info get [20] romio_filesystem_type: LUSTRE:
info get [21] romio_aggregator_list: 0
info get [22] striping_unit: 1048576
info get [23] striping_factor: 32
info get [24] romio_lustre_start_iodevice: 0
```

Miscellaneous file manipulation routines

- Pre-allocating space for a file (may be expensive)

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size);
```

- Resizing a file (may speed up first write to a file)

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size);
```

- Querying file size

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size);
```

- Querying file access mode

```
int MPI_File_get_amode(MPI_File fh, int *amode);
```

- File info object

```
int MPI_File_set_info(MPI_File fh, MPI_Info info);
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used);
```

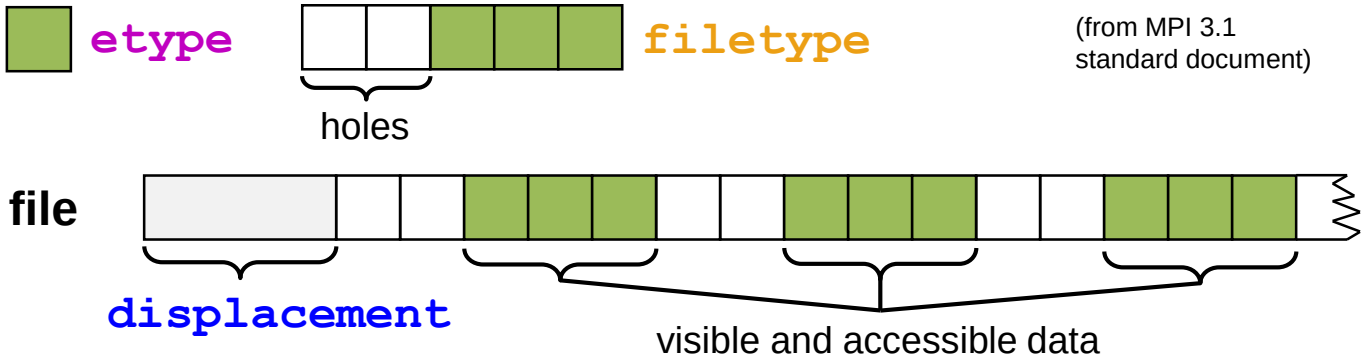
Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

File views

High Performance Computing

MPI I/O file views

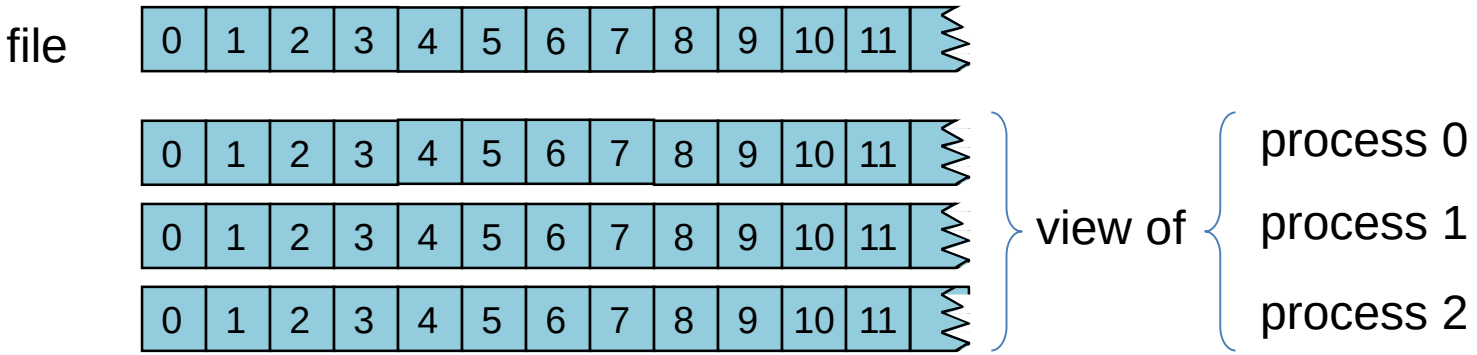


- Visible and accessible data from a file
- Each process has its own view
- View is described via (`displacement` , `etype` , `filetype`)
- Pattern of `filetype` is repeated beginning at `displacement`
- Views can be changed, but this is a collective operation
- Default view: linear byte stream (`0`, `MPI_BYTE` , `MPI_BYTE`)

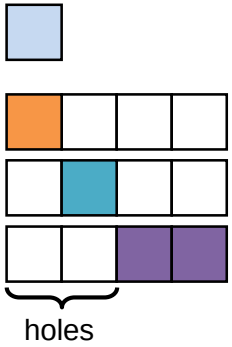
The default file view

- After file open, each file has the default view
- Default view: linear byte stream
  - `displacement = 0`
  - `etype = MPI_BYTE`
  - `filetype = MPI_BYTE`
- `MPI_BYTE` matches with any data type

`etype = filetype = MPI_BYTE`



A custom file view



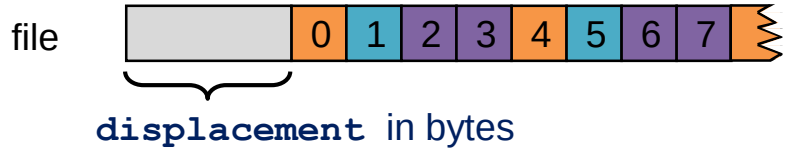
etype elementary datatype

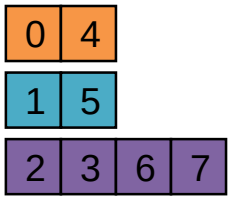
filetype process 0

filetype process 1

filetype process 2

tiling a file with filetypes:





view of process 0

view of process 1

view of process 2

example from MPI 3.1 standard document

Definitions

`file`	▪ ordered collection of data items
`displacement`	▪ position from the beginning of the file ▪ marks the start of the view, can be different on each process ▪ unit: byte
`etype`	▪ elementary data type ▪ unit of data access and positioning ▪ type displacements must be: nonnegative, monot. nondecreasing, and nonabsolute ▪ same for all processes
`filetype`	▪ single or multiple **`etypes`** ▪ size of holes must be multiples of **`etype`** extent ▪ repeated pattern after **`displacement`** ▪ type displacements must be: nonnegative, monot. nondecreasing, nonabsolute ▪ can be different for all processes
`view`	▪ accessible data of a file by a process ▪ defined by **`displacement, etype, filetype`**
`offset`	▪ position in file relative to current view ▪ type **`MPI_Offset`** in C, **`INTEGER(KIND=MPI_OFFSET_KIND)`** in Fortran ▪ unit: etype

Setting and getting the view

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
  MPI_Datatype etype, MPI_Datatype filetype,
  const char *datarep, MPI_Info info);
```

- Changes the process's view of the data
- Collective operation
- Local and shared file pointers are reset to zero
- `etype` and `filetype` must be committed types
- `datarep` is a string specifying the format data is written to a file:
  `native`, `internal`, `external32`, or user-defined (see next slide)
- Same `etype` extent and same `datarep` on all processes
- `disp`: `MPI_Offset` in C, `INTEGER(KIND=MPI_OFFSET_KIND)` in Fortran

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
  MPI_Datatype *etype, MPI_Datatype *filetype,  char *datarep);
```

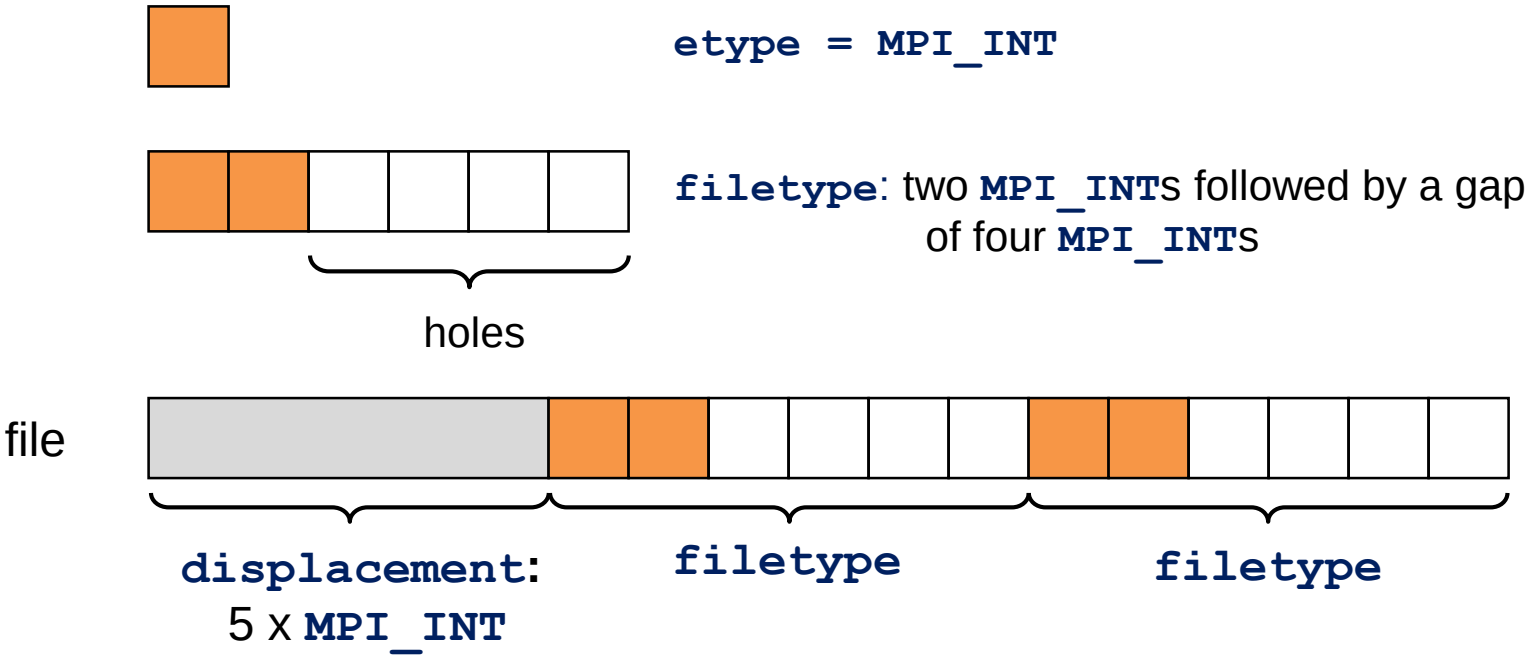
Data representations

native	▪ data stored in file identical to memory ▪ on homogeneous systems no loss in precision or I/O performance due to type conversions ▪ loss of interoperability on heterogeneous systems ▪ no guarantee that MPI files accessible from C/Fortran
internal	▪ data stored in implementation-specific format ▪ can be used with homogeneous or heterogeneous environments ▪ implementation will perform type conversions if necessary ▪ no guarantee that MPI files accessible from C/Fortran
external32	▪ follows standardized representation (big endian IEEE) ▪ all input/output operations are converted from/to **external32** ▪ files can be exported/imported between different MPI environments ▪ due to type conversions from (to) native to (from) **external32** data precision and I/O performance may be lost ▪ **internal** may be implemented as equal to **external32** ▪ can be read/written also by non-MPI programs

A simple file view example

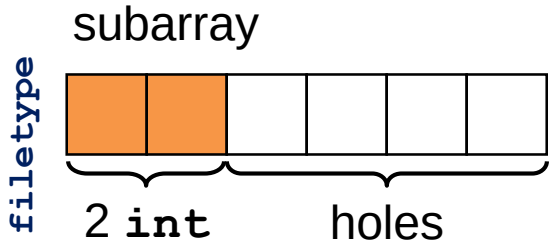
Basic example: File view for one process

- View contains holes with respect to original file



A simple file view example: C code

```
MPI_Offset disp;
MPI_Datatype etype, filetype;
int sizes[]      = { 6 };
int sub_sizes[]  = { 2 };
int start_idxs[] = { 0 };

MPI_Type_create_subarray(   1, sizes, sub_sizes, start_idxs,
                            MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(filetype);

disp  = 5 * 4;  // 4 = size of MPI_INT in bytes
etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR,
              MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);

MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```



Based on code by R. Thakur

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Reading and writing data

High Performance Computing

Reading and writing from/to files

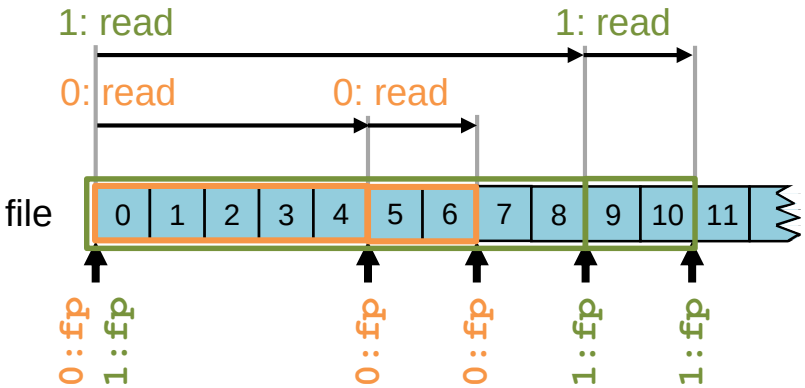
- Direction: Read / Write
- Positioning (realized via routine names)
  - explicit offset (**`_AT`**)
  - individual file pointer (no positional qualifier)
  - shared file pointer (**`_SHARED`** or **`_ORDERED`**)
    (different names used depending on whether non-collective or collective)
- Coordination
  - non-collective
  - collective (**`_ALL`**)
- Synchronization
  - blocking
  - non-blocking (**`_I`**…) and split collective (**`_BEGIN`**, **`_END`**)
- Atomicity (implemented with a separate API: **`MPI_File_set_atomicity`**)
  - atomic
  - non-atomic

All data access routines

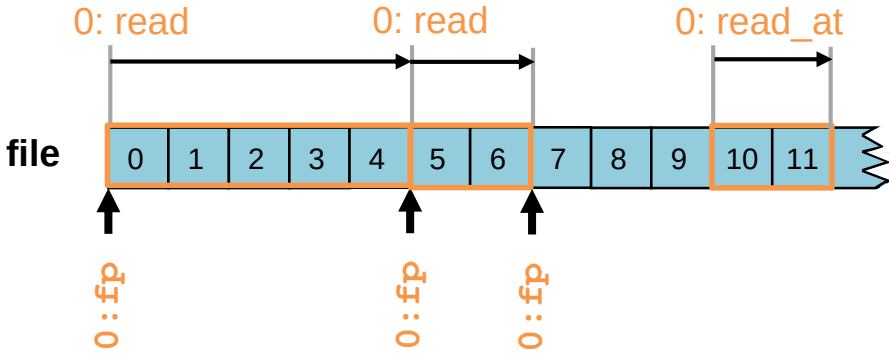
Positioning	Synchronization	Non-collective	Collective
Explicit offsets	blocking	`Reat_at`	`Read_at_all`
		`Write_at`	`Write_at_all`
	non-blocking	`Iread_at`	`Iread_at_all`
		`Iwrite_at`	`Iwrite_at_all`
	split collective		`Read_at_all_(begin|end)`
			`Write_at_all_(begin|end)`
Individual file pointers	blocking	`Read`	`Read_all`
		`Write`	`Write_all`
	non-blocking	`Iread`	`Iread_all`
		`Iwrite`	`Iwrite_all`
	split collective		`Read_all_(begin|end)`
			`Write_all_(begin|end)`
Shared file pointers	blocking	`Read_shared`	`Read_ordered`
		`Write_shared`	`Write_ordered`
	non-blocking	`Iread_shared`	
		`Iwrite_shared`	
	split collective		`Read_ordered_(begin|end)`
			`Write_ordered_(begin|end)`

Individual file pointers vs. explicit offsets

- Each process maintains its own individual file pointer:



- Explicit offsets do not affect file pointers



Explicit offsets

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
  void *buf, int count, MPI_Datatype datatype,
  MPI_Status *status);
```

- Read data starting at **offset**
- Read **count** elements of **datatype**
- Starting **offset** * units of **etype** from begin of view (**displacement**)
- Sequence of basic datatypes of **datatype** (= signature of **datatype**) must match contiguous copies of the **etype** of the current view
- EOF can be detected by noting that the amount of data read is less than **count**
  - i.e., EOF is no error
  - use `MPI_Get_count(&status, datatype, &recv_count);`
- Explicit offset routines do not alter file pointer

Individual file pointers

```
int MPI_File_read(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status);
```

- Arguments have same meaning as for **`MPI_File_reat_at`**
- **`offset`** is individual file pointer of calling process
- Individual file pointer is automatically incremented by

```
fp = fp + count * elements(datatype)/elements(etype)
```

- I.e., it points to the next **`etype`** after the last one that will be accessed (formula is not valid if EOF is reached)

- Behaves nearly like standard serial file I/O

Individual file pointers

- Set offset of individual file pointer fp:

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);
```

whence	description
MPI_SEEK_SET	set fp to offset
MPI_SEEK_CUR	set fp to fp + offset
MPI_SEEK_END	set fp to EOF + offset

- Get offset of individual file pointer:

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset);
```

- Get absolute byte position from offset for current view

```
int MPI_File_get_byte_offset(MPI_File fh,
        MPI_Offset offset, MPI_Offset *disp);
```

Shared file pointers

```
int MPI_File_read_shared(MPI_File fh,
    void *buf, int count, MPI_Datatype datatype,
    MPI_Status *status);
```

- One shared file pointer per **`MPI_File_open`**
- All processes must have the same view
- Individual file pointers are not affected
- Ordering during serialization is not deterministic
- Use **`*ordered`** (collective call) if determinism is required
- Use **`*shared`** routines to get/set file pointer

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Examples and use cases

High Performance Computing

Example: global matrix subarray

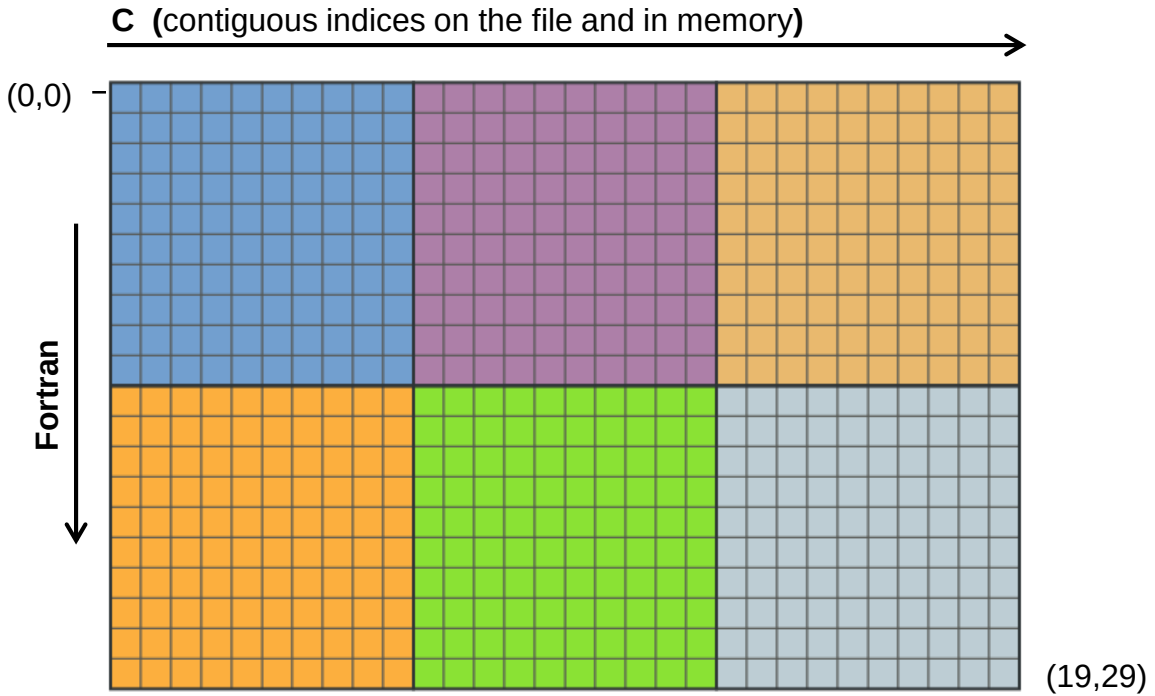
- Task
  - read a global matrix of size 20x30 from a file
  - store a subarray into a local array on each process
  - according to a given distribution scheme
- 2-dimensional distribution scheme: (BLOCK,BLOCK)
- **`larray`** = local array in each MPI process
  = subarray of the global array **`garray`**

- Remember: Contiguous index is language dependent
  - Fortran: `(1,1), (2,1), (3,1), ... , (1,10), (2,10), (3,10), ..., (20,30)`
  - C/C++: `[0][0], [0][1], [0][2], ... , [10][0], [10][1], [10][2], ..., [20][30]`

- same ordering on file (**`garray`**) and in memory (**`larray`**)

Global matrix subarray

- Process topology: 2x3
- global array on the file: 20x30
- distributed on local arrays in each processor: 10x10



Golbal matrix subarray

```
double larray[10][10];
MPI_Offset disp, offset, disp = 0, offset = 0;

ndims=2;
psizes[0]=2; period[0]=0;
psizes[1]=3; period[1]=0;
MPI_Cart_create(MPI_COMM_WORLD, ndims, psizes, period, 1, &comm);

MPI_Comm_rank(comm, &rank) ;
MPI_Cart_coords(comm, rank, ndims, coords);

gsizes[0]=20;  lsizes[0]=10;  starts[0]=coords[0]*lsizes[0];
gsizes[1]=30;  lsizes[1]=10;  starts[1]=coords[1]*lsizes[1];
MPI_Type_create_subarray(ndims, gsizes, lsizes, starts,
                         MPI_ORDER_C, MPI_DOUBLE, &stype);
MPI_Type_commit(&stype);

MPI_File_open(comm, file_name, MPI_MODE_READ, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_DOUBLE, stype, "native", MPI_INFO_NULL);
MPI_File_read_at_all(fh, offset,
                     larray, lsizes[0]*lsizes[1], MPI_DOUBLE,
                     &status);
```

Create virtual topology

Create custom datatype

Open file, create view, read data

Global matrix subarray

- All MPI coordinates and indices start with 0, even in Fortran (i.e., with **`MPI_ORDER_FORTRAN`**)
- MPI indices (here **`starts`**) may differ ( ) from Fortran indices
- Block distribution on 2*3 processes:

```
  rank   = 0                 rank   = 1                 rank   = 2
  coords = ( 0, 0)           coords = ( 0, 1)           coords = ( 0, 2)
  starts = ( 0, 0)           starts = ( 0,10)           starts = ( 0,20)
  garray(  0:9,  0:9)        garray(  0:9, 10:19)       garray(  0:9, 20:29)
= larray(  0:9,  0:9)      = larray ( 0:9,  0:9)      = larray(  0:9,  0:9)

  rank = 3                   rank = 4                   rank   = 5
  coords = ( 1, 0)           coords = ( 1, 1)           coords = ( 1, 2)
  starts = (10, 0)           starts = (10,10)           starts = (10,20)
  garray(10:19,  0:9)        garray(10:19, 10:19)       garray(10:19, 20:29)
= larray( 0:9,   0:9)      = larray( 0:9,   0:9)      = larray( 0:9,   0:9)
```

MPI I/O application scenarios I

- **Scenario A**: Each process has to read the whole file
- Solution 1: **`MPI_File_read_all`** — blocking

  collective with individual file pointers, with same view (**`displacement+etype+filetype`**) on all processes
- Solution 2: **`MPI_File_read_all_begin`** — nonblocking

  collective with individual file pointers, with same view (**`displacement+etype+filetype`**) on all processes, then computing some other initialization,

  `MPI_File_read_all_end`

MPI I/O application scenarios II

- **Scenario B**: The file contains a list of tasks, each task requires different compute time
- Solution: **`MPI_File_read_shared`**

  non-collective with a shared file pointer
  (same view is necessary for shared file pointer)

- **Scenario C**: The file contains a list of tasks, each task requires the same compute time
- Solution: **`MPI_File_read_ordered`**
  collective with a shared file pointer
  (same view is necessary for shared file pointer)
- or: **`MPI_File_read_all`**
  collective with individual file pointers,
  different views: **`filetype`** with
  `MPI_Type_create_subarray(…, &filetype)`

MPI I/O error handling

- File handles have their own error handler
- Default is **`MPI_ERRORS_RETURN`**, i.e., non-fatal
  - message passing: **`MPI_ERRORS_ARE_FATAL`**
- Default is associated with **`MPI_FILE_NULL`**
  - message passing: with **`MPI_COMM_WORLD`**

- Changing the default, e.g., after **`MPI_Init`**
  - `MPI_File_set_errhandler(MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL);`

- MPI is undefined after first erroneous MPI call,
- but a “high-quality implementation” will support I/O error handling facilities

MPI I/O summary

- Rich functionality provided to support various data representations and access options
- MPI I/O routines provide flexibility as well as portability
- Collective I/O routines can improve I/O performance
- Full implementation of MPI I/O available in all major implementations
  - Intel MPI
  - Open MPI
  - MVAPICH
  - …
- Generally, use of MPI I/O is often limited to special file systems; do not expect it to work on your average NFS-mounted `$HOME`
  - If it works at all, data loss might occur!